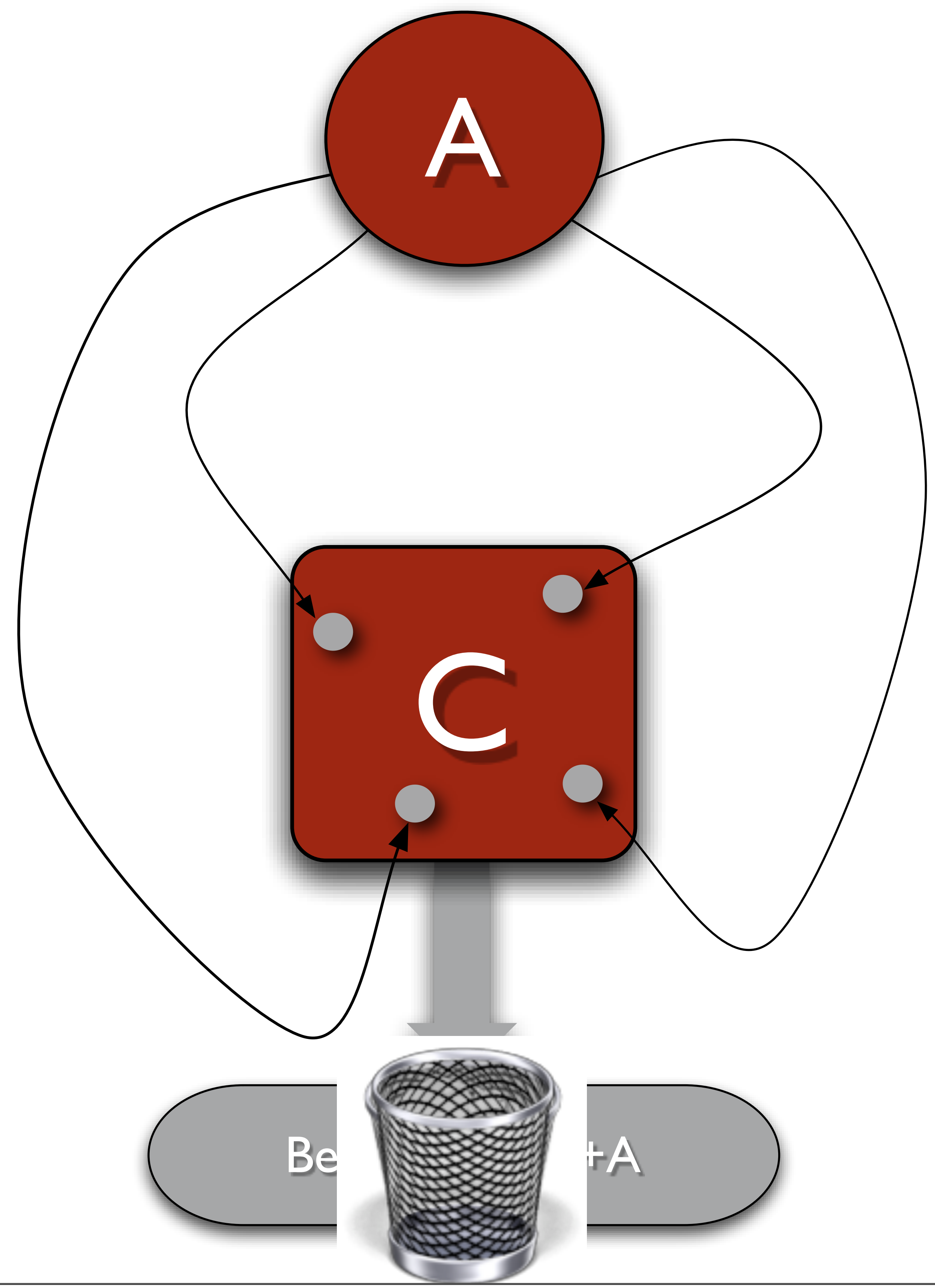
Enforcing Behavioral Constraints in Evolving Aspect-Oriented Programs

Raffi Khatchadourian[1*], Johan Dovland[2], and Neelam Soundarajan[1]

[1]The Ohio State University, USA
[2]University of Oslo, Norway
[*]Partially administered during visit to Lancaster University, UK

- AOP enables modular implementation of cross-cutting concerns.
- Both formal *and* informal reasoning about AOP presents unique challenges especially in respect to evolution.
- As components enter, exit, and re-enter software, conclusions about behavior of components may be invalidated.

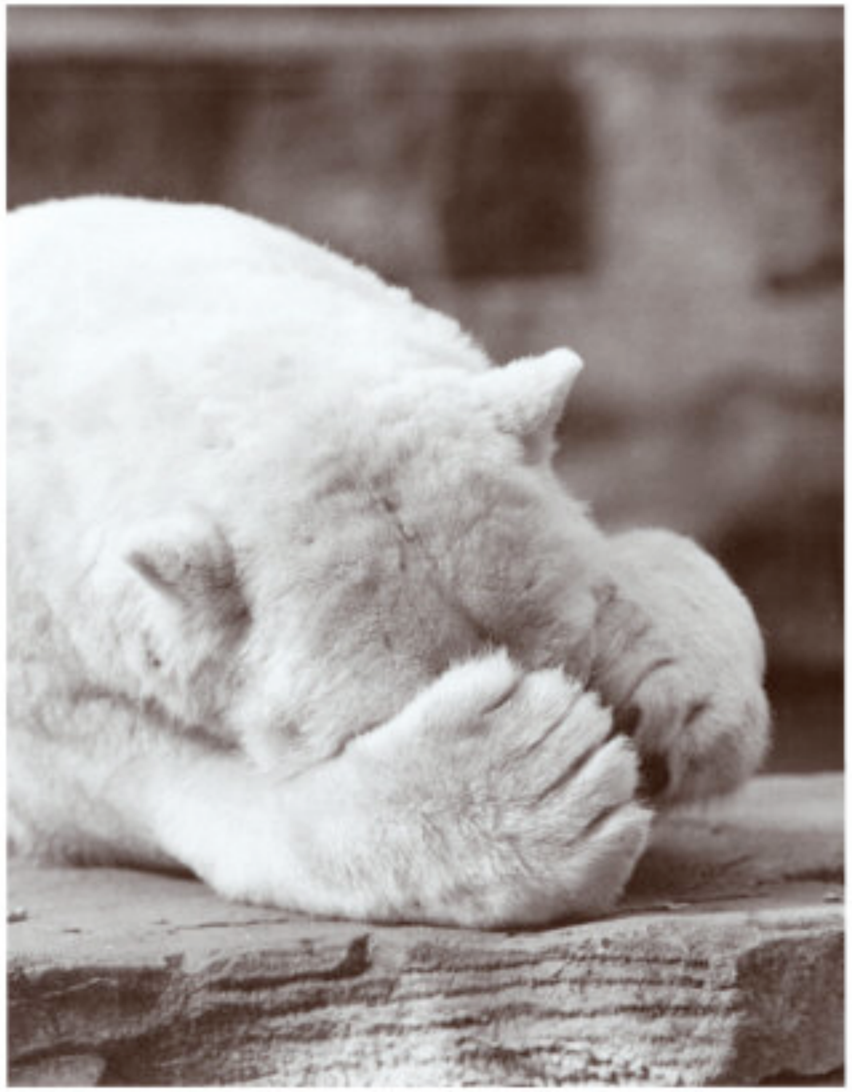

A
C
Be
+A

What We Want

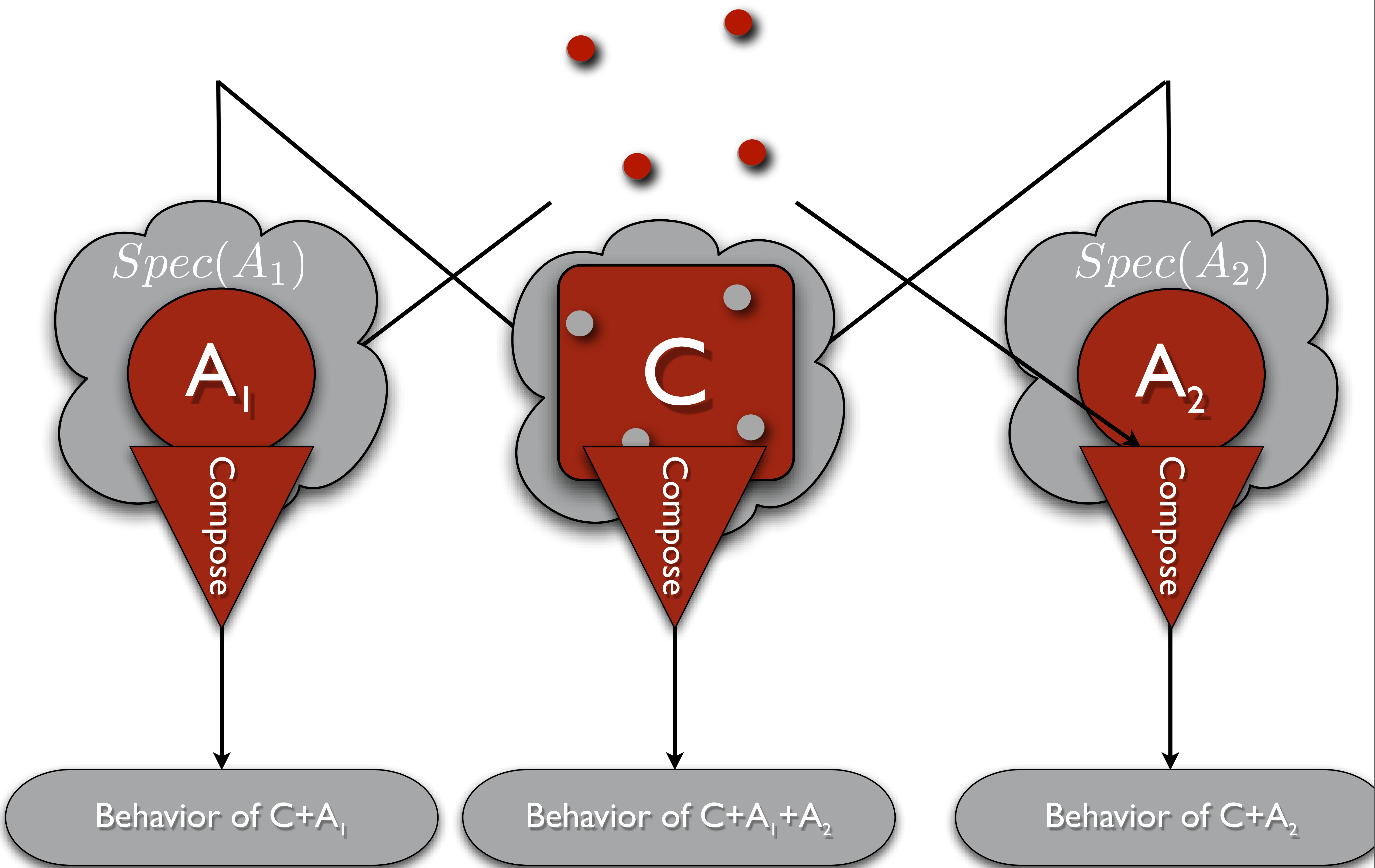
- Desire a *compositional* reasoning approach, however the invasive nature of AOP makes this difficult.
- In the worst case, changes made to a single component require reexamining the entire program.

- Can we draw meaningful conclusions about component code *without* considering the *actual* advice code?
- Can we specify the behavior of components without any particular advice in mind?
- Can we *parameterize* specifications over *all* possibly applicable aspects?
- Can we suitably constrain the behavior of aspects as the software evolves?

- Using interface is one answer (e.g., XPIs, Open Modules)
- But it would be nice to have a way to derive the *enriched* behavior of the base plus the aspects at compile time.



- AO programs inherently enjoy *plug-n-play* capabilities [Laddad03]
- Crosscutting features can be *plugged-in* to *enrich* the behavior of advised components.
- Likewise, can we specify components so that we can derive their behaviors in a similar fashion?


$Spec(A_1)$
A1
Compose
C
Compose
$Spec(A_2)$
A2
Compose
Behavior of C+A1
Behavior of C+A1+A2
Behavior of C+A2

Obstacles

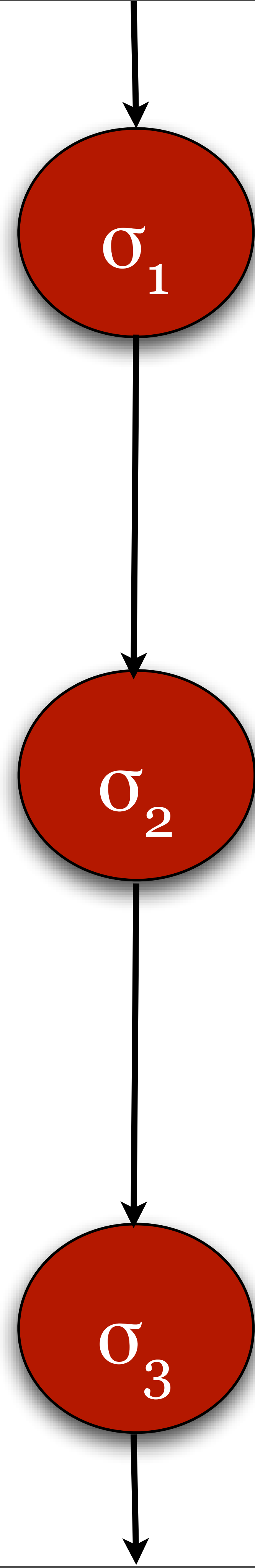
- ***Usefulness***
  - Is it possible to draw meaningful conclusions from such incomplete information?
- ***Obliviousness***
  - Specifications contain "slots" for applications of crosscutting concerns.

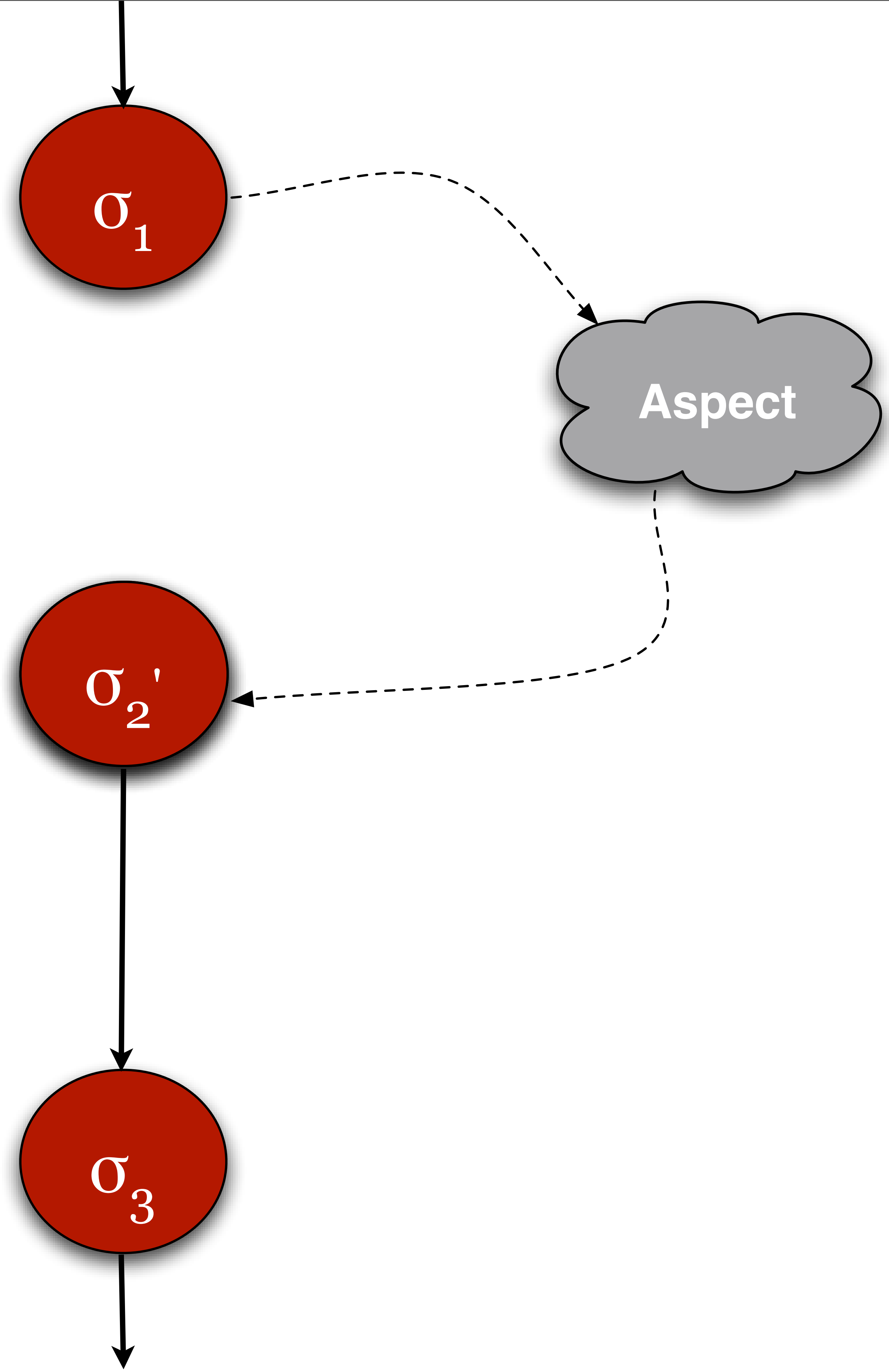
- ***Abstraction***
  - Competing forces:
    - Specs abstract internal details components, aspects directly manipulate them.
- ***Composition***
  - Which pegs go into which holes?
  - How to deal with dynamic and lexical pointcuts?
- ***Complexity***
  - What if no advice is applicable?

Tackling Specification Complexity

- May need to make assumptions about the behavior of evolving components.
- *Specification pointcuts*
  - *Pointcut interfaces* [Gudmundson01] annotated with behavioral specifications.
  - "Exported" internal semantic events within the component.
  - Adopt a *rely-guarantee* approach [Xu97] from concurrent programming to constrain the behavior of all possibly applicable advice using a *rely* clause.
  - A *guar* clause may be used to constrain components.

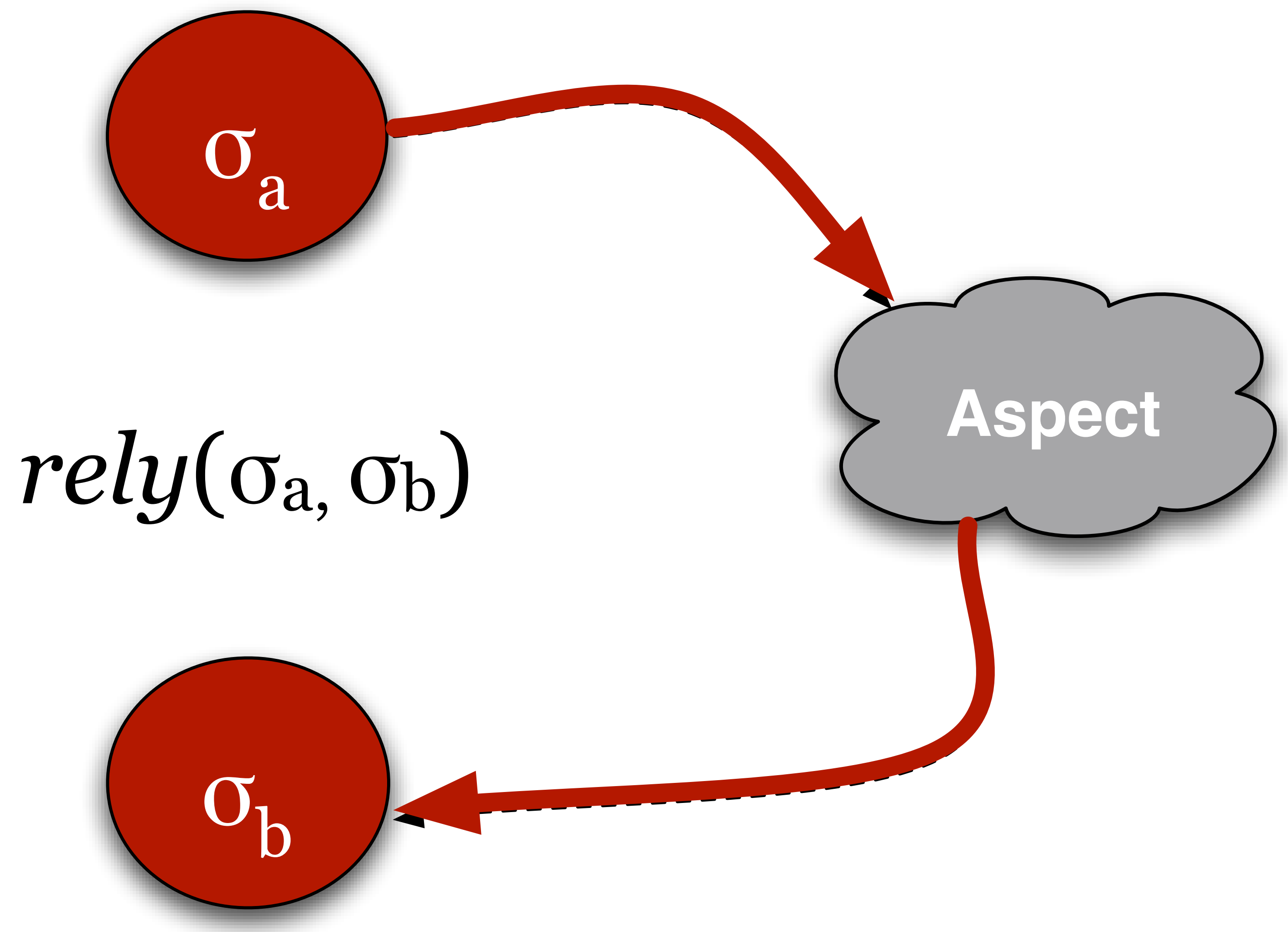
σ	the set of all variables of the program
$\sigma_i, \sigma_j, \ldots$	states in which each variable has a particular value


σ1
σ2
σ3



σ_1
Aspect
σ_2'
σ_3

The state at a point in the execution of a component is σ_a.



The state when the class gets control back from an aspect is σ_b.

This is "Harmless"[D&W POPL'06]

entire state of C

$$rely(\sigma, \sigma') \equiv (\sigma = \sigma')$$

Forbids any applicable advice from making any changes in the state!

Deriving Effective Behavior

- Constraining parameterized behavior reduces complexity, but ...
  - How are *formal* parameters expressed?
  - How are *actual* parameters deduced?
  - How are the specifications *composed*?
- Aspects are typically used to *enrich* the behavior of the an underlying component.
- Thus, we want to deriving the *actual* behavior of components with the aspects.

Join Point Traces

- A *Join Point Trace* (JPT) variable is introduced to track the *flow-of-control* through various join points within a component.
- A JPT is used as a parameter over the actions of all possibly applicable aspects.
- Method post-conditions will references to the JPT.
- Informally, a JPT is used to refer to the actions and resulting values taken by advice at certain join point.

- The JPT is composed of several components that are associated with each join point.
- Just as there are different *kinds* of join points (e.g., `call`, `execution`), there different kinds of JPT entries.

State Vectors

$$(oid, mid, aid, args, res, \sigma, \sigma')$$

Called Method

Argument Values

Called Object

Applicable Aspect

Method Return Value

$\sigma[oid]$ State of object `oid` after completion of method `mid`

$\sigma'[oid]$ State of object `oid` after completion of aspect `aid`

No applicable advice $\implies \sigma = \sigma'$

Rule for method specification

Normal pre-condition

$$\texttt{C.m} :: \langle pre,\ post,\ guar(),\ rely()\ \rangle$$

Post-condition, may include references to *portions* of JPT

R/G Clauses

Rule for method specification

$$pre \wedge [\lambda\tau = \langle(inv, \texttt{C.m})\rangle] \Rightarrow p$$

Invocation of `C.m` on the *local* JPT

$$\frac{pre \wedge [\lambda\tau = \langle(inv, \texttt{C.m})\rangle] \Rightarrow p}{\texttt{C.m} :: \langle pre,\ post,\ guar(),\ rely()\ \rangle}$$

Rule for method specification

$$\frac{\begin{array}{c} pre \wedge [\lambda\tau = \langle(inv, \mathtt{C.m})\rangle] \Rightarrow p \\ \{p\}S\{q\} \end{array}}{\mathtt{C.m} :: \langle pre,\ post,\ guar(),\ rely()\ \rangle}$$

Classic Hoare Triple

Rule for method specification

$$pre \wedge [\lambda\tau = \langle(inv, \texttt{C.m})\rangle] \Rightarrow p$$

$$\{p\}S\{q\}$$

$$q \Rightarrow guar()$$

Don't forget about the guarantee

$$, \ guar(), \ rely() \ \rangle$$

Rule for method specification

$$pre \wedge [\lambda\tau = \langle(inv, \texttt{C.m})\rangle] \Rightarrow p$$

$$\{p\}S\{q\}$$

$$q \Rightarrow guar()$$

$$[q \wedge rely(\sigma, \sigma')] \Rightarrow$$

$$\texttt{C.m} :: \langle pre,\ post,$$

If when q holds and applicable advice behaves properly implies that ...

Rule for method specification

$\langle (inv, \texttt{C.m}) \rangle]$

$S\{q\}$

$guar()$

$(\sigma, \sigma')] \Rightarrow$

$$post[\lambda\tau \leftarrow \lambda\tau^\frown(this, C.m, ?, args, res, \sigma, \sigma'),\ \sigma \leftarrow \sigma']$$

$\texttt{C}$ … $ar(),\ rely()\ \rangle$

Not sure which aspect is applicable yet, so we'll leave this blank

Replace all occurrences of σ with σ'

... our post-condition holds with a a *new* entry in the local JPT

$$\{ p \} \; \texttt{ob.m(args)} \; \{ q \}$$

Rule for Method Calls

Substitute actuals for formals

$$\frac{p \Rightarrow \texttt{C.m.}\,pre[pars\,/\,args]}{\{\, p \,\}\ \texttt{ob.m(args)}\ \{\, q \,\}}$$

$$p \Rightarrow \texttt{C.m.}pre[p$$

$$\texttt{C.m.}post \Rightarrow q[\lambda\tau 1 / \lambda\tau 1 \hat{} \lambda\tau 2,\ args / pars]$$

$$\{ p \}$$

Local JPT for *caller*

Local J... *calle*...

Substitute formals for actuals

Rule for Aspect Application (Simple)

Base-code pre-condition

Aspect post-condition

$$\{pre \wedge ap\}\ \texttt{C.m}() + \texttt{A}\ \{post \wedge aq\}$$

Aspect pre-condition

Base-code post-condition

Rule for Aspect Application (Simple)

Base-code satisfies *guar*

Advice body

$$\{\ guar(\sigma) \wedge ap\ \}\ \mathtt{A}_{adv}\ \{rely[\sigma/\sigma@pre, \sigma'/\sigma] \wedge\ aq\ \}$$

$$\{pre \quad\quad post \wedge aq\ \}$$

State vector immediately prior to the execution of the advice

Rule for Aspect Application (Simple)

$$\frac{\{\, guar(\sigma) \wedge ap \,\}\ \mathtt{A}_{adv}\ \{rely[\sigma/\sigma@pre, \sigma'/\sigma] \wedge\ aq\,\} \quad \mathtt{C.m} :: \langle pre,\ post,\ guar,\ rely\,\rangle}{\{pre \wedge ap\,\}\ \mathtt{C.m}() + \mathtt{A}\ \{post \wedge aq\,\}}$$

- On-going work (hopefully thesis worthy! ;))
- Complete formal model (suggestions here?)
- Sound axiomatic proof system
- Curbing notational complexity via predicates.
- Integration with IDE/theorem provers.
  - Complement the Eclipse AJDT with a *behavioral* cross reference view?
- Integration with languages (e.g., via annotated pointcuts, JML)